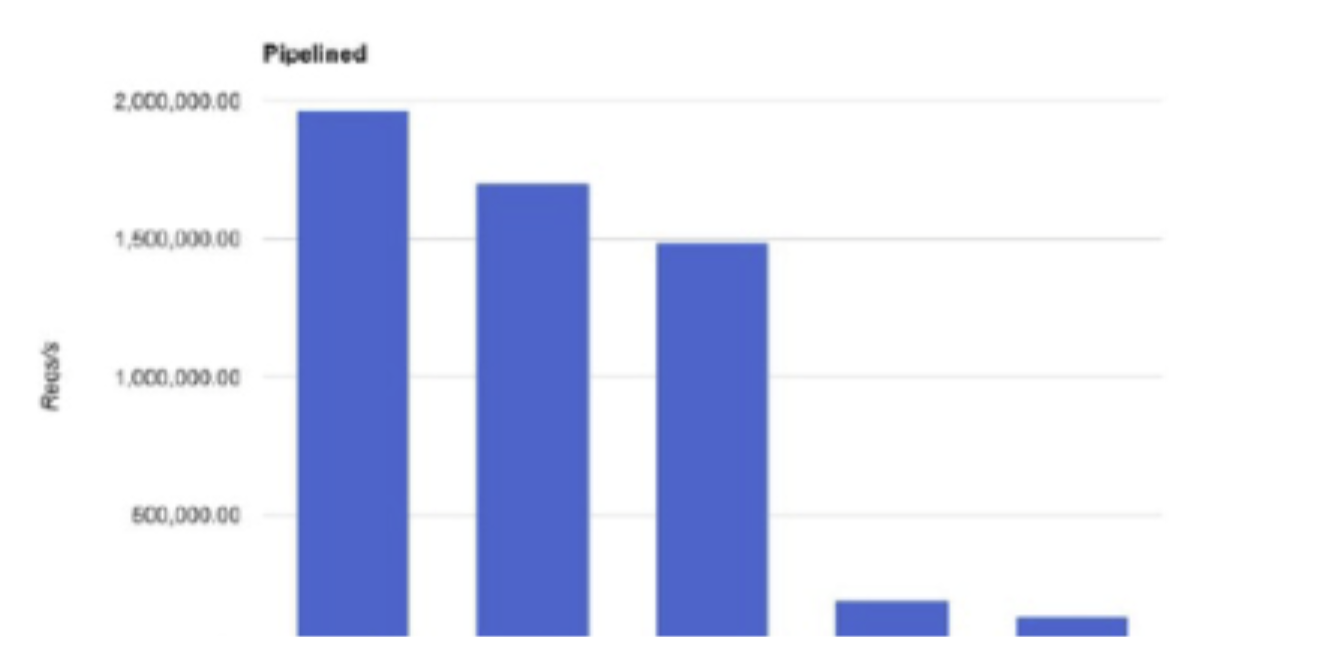
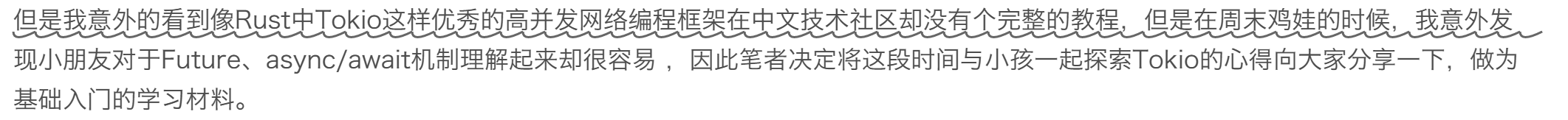
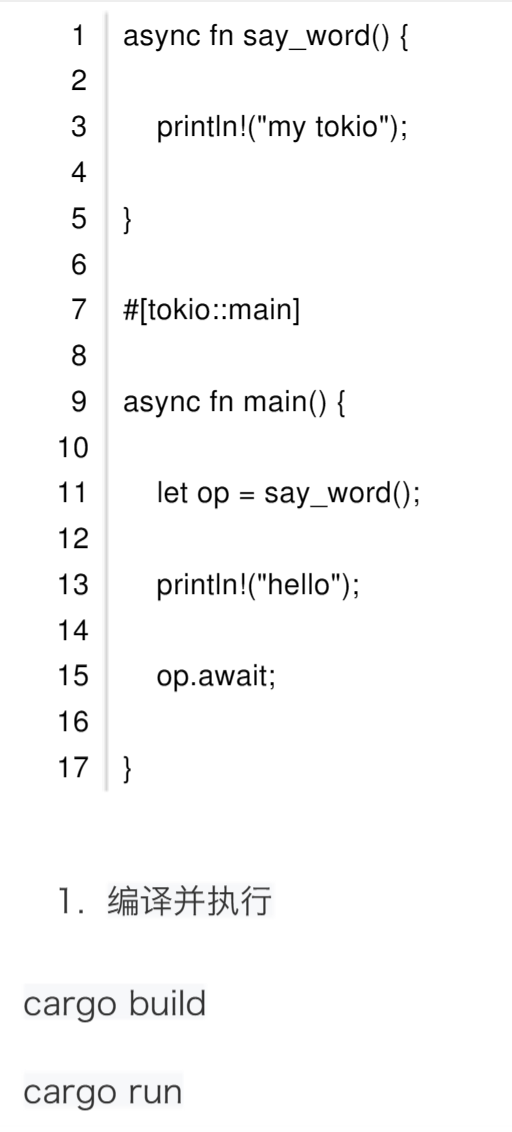
# 

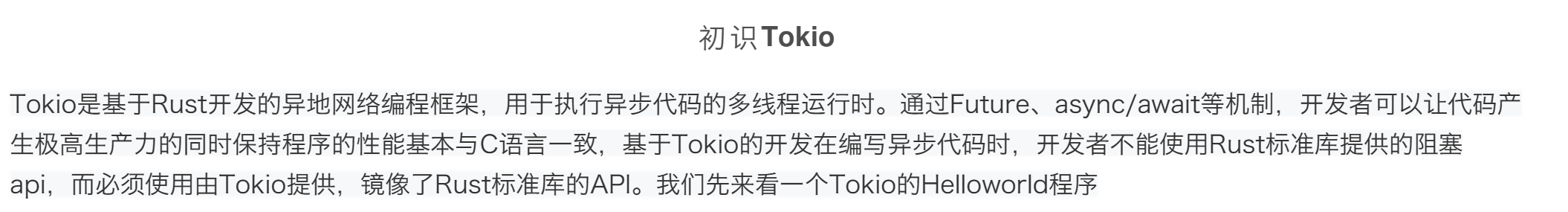












# Rust网络编程框架-Tokio进阶

 2022-08-28 21:41 116阅读  [0赞](https://2fwww.dandelioncloud.cn/article/details/1563884471466553346" \l "comments)

我们在上文《小朋友也能听懂的Rust网络编程框架知识-Tokio基础篇》对于Tokio的基础知识进行了一下初步的介绍，本文就对于Tokio的用法及原理进行进一步的介绍与说明。

目前市面上绝大多数编程语言所编写的程序，执行程序与代码编写顺序完全相同，当然有的读者可能会提到CPU的乱序执行机制，但乱序执行从本质上讲还是顺序提交的，程序在第一行执行完成之后再去执行下一行，并以此类推，是通用的编程模式。

在这种传统的式编程范式中，当程序遇到耗时操作时，会一直阻塞直到操作完成。比如建立TCP连接可能需要与网络上的对端节点进行若干次握手，这可能会花费相当多的时间。在此期间，线程被阻塞而无法完成其它操作。

在传统的编程范式中往往使用回调机制来进行资源调配的优化，对于不能立即完成的操作将被挂起到后台，这种情况下线程不会被阻塞，可以继续执行其它任务。一旦操作完成，该任务的回调函数将被调用，从而使任务最终完成。尽管回调模式可以带来使应用程序的效率更高，但也会导致程序更复杂。开发者需要跟踪异步操作完成后恢复工作所需的所有状态，从我的经验来看，这是一项特别乏味而且极容易出错的工作任务。

### ****为什么需要异步调用****

以下例程部分依赖于mini-redis模块在执行了cargo install mini-redis之后，并在Cargo.toml最后加入以下配置项之后，

1. tokio = { version = "1", features = ["full"] }
2. mini-redis = "0.4"

即可顺利执行下列代码：

1. use tokio::net::{TcpListener, TcpStream};
2. use mini\_redis::{Connection, Frame};
3. #[tokio::main]
4. async fn main() {
5. // 绑定端口
6. let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
7. loop {
8. // 监控端口消息，对于每个socket请求，都启动一个folk进程，进行处理
9. let (socket, \_) = listener.accept().await.unwrap();
10. Process(socket).await;
11. }
12. }
13. async fn Process(socket: TcpStream) {
14. let mut connection = Connection::new(socket);
15. if let Some(frame) = connection.read\_frame().await.unwrap() {
16. println!("WE GOT: {:?}", frame);
17. let response = Frame::Error("not finished".to\_string());
18. connection.write\_frame(&response).await.unwrap();
19. }
20. }

以上代码可能我们在其它语言编程中所经常遇到的，对于每个Socket连接都通过一个线程来处理（当然这里只是以Rust为例说明，在Tokio中不推荐这种做法，我也就没有另行启动线程）并且最关键的一点是process(socket).await;是同步调用，也就是说在线程阻塞在process函数时并没有其它事情可做，整个线程必须要等到响应被完全写入socket stream才能返回。而这种并发处理与我们尽可能多的同时处理更多请求的初衷是不一致的。

这里笔者必须要指出，并发和并行完全是两件事。多个任务交替执行是并发，并行是有多个人，一个人负责一个任务。而Rust的Tokio最大就是并发效率很高，线程并不需要去等待那些无效的任务，众多并发任务之间由Tokio去统一调度。

### ****Tokio的答案****

Rust使用spawn关键字来建立此类并发任务的任务池，按照笔者的理解，这和线程池不是一个概念，因为并发的任务可能有多个线程共同处理，也可能只有一个线程就搞定了。在使用Rust这种并发任务的异步函数使用async关键字修饰，在异步函数的函数体内任何类似于await的阻塞调用用都会使任务将控制权交还给线程。当操作进程在后台时，线程可以做其他工作。操作产生的结果也将形成一个Future，也就是未来才会产生的值被系统以变通的方式优化处理，改写后的代码如下：

1. use tokio::net::{TcpListener, TcpStream};
2. use mini\_redis::{Connection, Frame};
3. #[tokio::main]
4. async fn main() {
5. // 绑定端口
6. let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
7. loop {
8. // 监控端口消息，对于每个socket请求，都启动一个folk进程，进行处理
9. let (socket, \_) = listener.accept().await.unwrap();
10. tokio::spawn(async move {
11. process(socket).await;
12. });
13. }
14. }
15. async fn Process(socket: TcpStream) {
16. let mut connection = Connection::new(socket);
17. if let Some(frame) = connection.read\_frame().await.unwrap() {
18. println!("WE GOT: {:?}", frame);
19. let response = Frame::Error("not finished".to\_string());
20. connection.write\_frame(&response).await.unwrap();
21. }
22. }

Tokio的任务通过tokio::spawn来创建，spawn函数返回一个JoinHandle，调用者可以使用JoinHandle它与Tokio的任务进行交互。async修饰的函数的返回值以Future方式返回。调用者可以使用.awai来Future的执行结果。代码如下：

1. #[tokio::main]async fn main() {
2. let handle = tokio::spawn(async {
3. "hello beyondma"
4. });
5. let out = handle.await.unwrap();
6. println!("GOT {}", out);
7. }

上述程序运行结果为

GOT hello beyondma

当Tokio任务执行过程中遇到错误时，JoinHandle将返回一个Err。当任务失败时，或者当任务被强制关闭时，是铁定会返回ERR的。Tokio任务由Tokio调度器管理的最小可执行单元。正如上文所说Tokio的任务可能在同一个线程上执行，也可能在不同的线程上执行，这种多路复用机制可以参考上文《《小朋友也能听懂的Rust网络编程框架知识-Tokio基础篇》》

### **Tokio任务之间的同步与通信**

我们知道Rust有着比较独特的变量生命周期机制，在之前的示例代码当中都是用了move关键字来强制传递变量所属关系的，如下：

tokio::spawn(async move {

process(socket).await;

});

那么如何在各个Tokio任务之间进行通信与状态同步也是个值得在本文中讨论的问题。

这里我们先来讨论比较简单的情况，可以用Arc<mutex>类型，也就是加互斥锁的哈希表来进行任务间的信息传递与同步，使用clone方法来为每个任务获取自己的哈希表实例。具体如下：

1. use tokio::net::TcpListener;
2. use std::collections::HashMap;
3. use std::sync::{Arc, Mutex};
4. #[tokio::main]
5. async fn main() {
6. let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
7. println!("Listening");
8. let hashMap= Arc::new(Mutex::new(HashMap::new()));
9. loop {
10. let (socket, \_) = listener.accept().await.unwrap();
11. // Clone the handle to the hash map.
12. let thisMap = hashMap.clone();
13. println!("Accepted");
14. tokio::spawn(async move {
15. let mut thisMap = thisMap .lock().unwrap();
16. thisMap .insert("hello", "beyondma");
17. println!("{:?}",thisMap );
18. });
19. }
20. }

这样随便找其它终端或者在本机上执行telnet 服务器IP 6379

就可以看到以下结果

1. Listening
2. Accepted
3. {"hello": "beyondma"}

这里这个hashMap的确可以在进程之间进行信息的共享与同步，但是在这种高并发的框架中一般还是推荐使用管道（channel)来进行相关操作。有关channel的话题我们会在下次再深入讲解。

Tokio的任务非常轻,只需要一个64字节的上下文即可，考虑到Rust中也没有GC机制，因此基于Tokio理论上完全可以做出比Golang支持更多并发的应用程序，这也是笔者会计划用3篇左右的系列文章来对于Tokio进行详细介绍的原因。